Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automation

Laboratory Work No. 1

User Interaction: STDIO — Serial Interface

Discipline: *Embedded Systems*

Student: *Cebotari Alexandru*
Group: *FAF-233*
Supervisor: *Martiniuc Alexei*

Chişinău, 2026

Contents

1 Domain Analysis

1.1 Objective

The objective of this laboratory work is to become familiar with the **STDIO** (Standard Input/Output) library for serial communication on an embedded system and to implement a simple application that controls an **LED** through text commands transmitted from a **serial terminal**.

The specific goals are:

- Understanding the fundamentals of serial (UART) communication.
- Using the STDIO library (`printf`, `scanf`, etc.) for text-based data exchange instead of Arduino-specific `Serial.print()` functions.
- Designing an application that interprets commands received over the serial interface.
- Developing a **modular** solution where each peripheral has its own dedicated source files for future reusability.

1.2 Technologies and Context

1.2.1 UART Serial Communication

UART (Universal Asynchronous Receiver-Transmitter) is one of the simplest and most widely used serial communication protocols in embedded systems. It uses two data lines — **TX** (transmit) and **RX** (receive) — to exchange data asynchronously at an agreed-upon baud rate. On the Arduino Uno, the ATmega328P microcontroller has a built-in USART peripheral connected to digital pins 0 (RX) and 1 (TX), which is also routed through a USB-to-serial converter chip (typically CH340 or ATmega16U2) so that the MCU can communicate with a PC over USB.

In this project, the baud rate is configured to **9600 bps**, which is a widely supported default for serial monitors.

1.2.2 STDIO Library on AVR

On desktop systems, `printf()` and `scanf()` write to and read from the console. On an AVR microcontroller such as the ATmega328P, there is no console—`stdout` and `stdin` are `NULL` by default. The AVR `libc` provides the function `fdev_setup_stream()` which allows the programmer to **redirect** these standard streams to any custom character I/O functions.

In this project, instead of using the Arduino-specific `Serial.println()`, we redirect `stdout` and `stdin` to the UART by providing custom `uartPutChar()` and `uartGetChar()` callback functions. This means all subsequent calls to `printf()` send data over the serial port, and input can be read via standard C functions. This approach was chosen specifically to satisfy the laboratory requirement of using the STDIO library.

1.2.3 Why STDIO Instead of Serial.print()?

- **Portability:** Code written with `printf`/`scanf` can be reused across different platforms (AVR, ARM, x86) with minimal changes to the I/O redirection layer.

- **Formatted output:** `printf()` offers powerful formatting capabilities (e.g., `%d`, `%s`, `%x`) in a single function call.
- **Educational value:** Understanding how low-level stream redirection works deepens knowledge of the hardware-software interface stack.

1.3 Hardware Components

Table 1: Bill of Materials

#	Component	Specification	Role
1	Arduino Uno R3	ATmega328P, 16 MHz	Main MCU board
2	USB Type-B cable	—	Power + serial communication
3	LED	Standard 5 mm, red	Visual output indicator
4	Resistor	220 Ω	Current limiting for LED
5	Breadboard	400-tie point	Prototyping platform
6	Jumper wires	Male-to-male	Circuit connections

Note: For initial testing, the **built-in LED** on pin 13 of the Arduino Uno was used, which does not require an external resistor or breadboard. For a standalone demonstration, an external LED with a 220 Ω resistor is connected between pin 13 and GND.

1.4 Detailed Hardware Component Descriptions

1.4.1 Arduino Uno R3 (ATmega328P Microcontroller)

The Arduino Uno R3 is a development board built around the **ATmega328P** 8-bit AVR microcontroller from Microchip (formerly Atmel). Key specifications:

- **Clock frequency:** 16 MHz (external crystal oscillator).
- **Flash memory:** 32 KB (0.5 KB used by bootloader).
- **SRAM:** 2 KB — used for variables, stack, and the command buffer in our application.
- **Digital I/O pins:** 14 (6 with PWM output), of which pin 13 is connected to the on-board LED.
- **USART:** 1 hardware serial port (pins 0/1), used in this project for STDIO communication at 9600 baud.
- **USB interface:** A CH340 or ATmega16U2 chip converts USB to TTL-level UART, enabling both programming (firmware upload) and serial data exchange with the PC over a single USB cable.

In this project, the Arduino Uno serves as the central processing unit that receives serial commands, processes them, and drives the LED accordingly.

1.4.2 LED (Light-Emitting Diode)

An LED is a semiconductor device that emits light when current flows through it in the forward direction. Key characteristics:

- **Forward voltage (V_f):** approximately 1.8–2.2 V for a standard red LED.

- **Maximum forward current:** typically 20 mA.
- **Polarity:** The longer leg is the anode (+), the shorter leg is the cathode (–).

In this project, the LED serves as the **visual output indicator**—it is the peripheral being controlled by serial commands. When the user sends "led on", the GPIO pin is set HIGH (5 V), causing current to flow through the LED. When "led off" is sent, the pin is set LOW (0 V) and the LED turns off.

1.4.3 Resistor (220 Ω)

The resistor is connected in series with the LED to **limit the current** and prevent damage to both the LED and the MCU's GPIO pin. Without it, the LED would draw excessive current and potentially burn out. The resistor value is calculated using Ohm's law:

$$I = \frac{V_{pin} - V_f}{R} = \frac{5\text{V} - 2\text{V}}{220\,\Omega} \approx 13.6\,\text{mA}$$

This is safely below the ATmega328P's maximum GPIO output of 20 mA and within the LED's rated current.

1.4.4 Breadboard and Jumper Wires

The **breadboard** is a solderless prototyping platform with internal bus connections that allow components to be connected without soldering. **Jumper wires** (male-to-male) are used to connect the Arduino's pin 13 and GND to the breadboard rows where the resistor and LED are placed. These components are only needed when using an external LED; for the built-in LED on pin 13, no breadboard is required.

1.4.5 USB Cable (Type-B)

The USB cable serves a dual purpose:

- **Power supply:** Provides 5 V from the PC's USB port to power the Arduino and all connected components.
- **Serial communication:** The USB-to-UART bridge chip on the Arduino converts USB data to TTL-level serial signals. This is the physical transport layer over which all `printf()` output and terminal input travel between the MCU and the PC's serial monitor.

1.5 Hardware–Software Peripheral Stack

The following table describes the **peripheral interface stack**—how each hardware peripheral is accessed from the software layer, from the highest abstraction down to the physical hardware:

Table 2: Hardware–Software Peripheral Stack

Layer	LED Peripheral	Serial (UART) Peripheral
Application	`serialCommandProcess()` calls `ledOn()`/`ledOff()`	User types in serial monitor and reads confirmation messages
Module API	`led.h`: `ledInit()`, `ledOn()`, `ledOff()`	`serialCommand.h`: `serialCommandInit()`, `serialCommandRead()`, `serialCommandProcess()`
STDIO Layer	— (direct GPIO)	`printf()` → `uartPutChar()` `stdin` → `uartGetChar()`
Arduino HAL	`pinMode()`, `digitalWrite()`	`Serial.begin()`, `Serial.write()`, `Serial.read()`
AVR Registers	`DDRB`, `PORTB` (Port B, bit 5 for pin 13)	`UBRR0`, `UCSR0B`, `UDR0` (USART0 control/data registers)
Physical HW	GPIO Pin 13 → Resistor → LED → GND	USART → USB-UART bridge → USB cable → PC

Key observations about the HW–SW interface:

- The **LED peripheral** has a short stack: the application calls module functions (`ledOn`), which directly use Arduino HAL functions (`digitalWrite`), which in turn manipulate AVR GPIO registers (`PORTB`).
- The **Serial peripheral** has an additional **STDIO abstraction layer** between the module and the HAL. The `fdev_setup_stream()` function binds the C standard library's `printf`/`scanf` to custom UART callbacks. This extra layer is what allows us to use portable `printf()` instead of Arduino-specific `Serial.print()`.
- Both peripherals ultimately communicate through **memory-mapped I/O registers** on the ATmega328P. The Arduino HAL abstracts these registers into simple function calls, and our modules further abstract the HAL into domain-specific operations ("turn LED on" rather than "set bit 5 of PORTB").

1.6 Software Tools

Table 3: Software Environment

Tool	Version / Detail	Purpose
Visual Studio Code	Latest	Source code editor (IDE)
PlatformIO IDE Extension	Latest	Build system, upload, serial monitor
PlatformIO Core (CLI)	v6.x	`pio run`, `pio device monitor`
Arduino Framework	AVR platform	HAL for ATmega328P
AVR-GCC + AVR libc	Bundled with PlatformIO	Compiler toolchain + STDIO support
Serial Monitor	PlatformIO built-in	Terminal for sending commands

PlatformIO was chosen instead of the standard Arduino IDE because it integrates directly with VS Code, supports multi-file projects natively, provides a proper build system with de-

pendency management, and allows working with `include/` and `src/` directories—essential for the modular architecture required by this laboratory.

2 Design

2.1 System Architecture

The system follows a **layered modular architecture** with clear separation of concerns. The diagram below illustrates how the components interact:

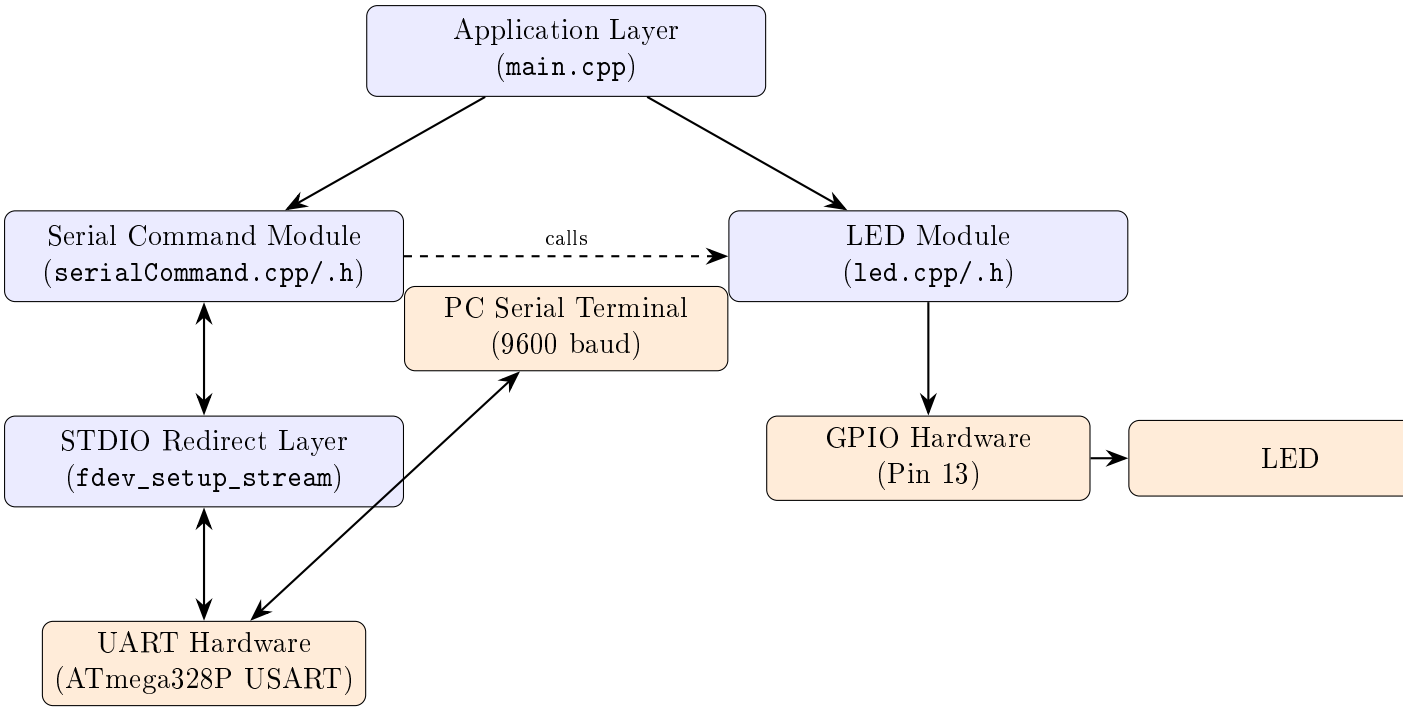

Figure 1: System Architecture Diagram

Component roles:

- **Application Layer** (`main.cpp`): Initializes all modules and runs the main loop. It delegates serial reading and command processing to the Serial Command Module.
- **Serial Command Module** (`serialCommand.cpp/h`): Handles STDIO stream redirection (mapping `stdout`/`stdin` to UART), reads characters from the serial buffer, assembles them into complete command strings, and dispatches recognized commands to the appropriate peripheral module.
- **LED Module** (`led.cpp/h`): Provides a clean API for LED control (`ledInit`, `ledOn`, `ledOff`, `ledGetState`). Encapsulates all GPIO pin manipulation so that the rest of the application never directly calls `digitalWrite()`.
- **STDIO Redirect Layer**: Uses AVR libc's `fdev_setup_stream()` to bind `printf()`/`scanf()` to the UART. Custom `uartPutChar()` and `uartGetChar()` functions serve as the bridge between the C standard library and the hardware serial peripheral.
- **UART Hardware**: The ATmega328P's built-in USART peripheral, accessed via the Arduino `Serial` object, which is initialized at 9600 baud.
- **GPIO Hardware**: Digital pin 13 configured as OUTPUT to drive the LED.

2.2 LED Module Flowchart

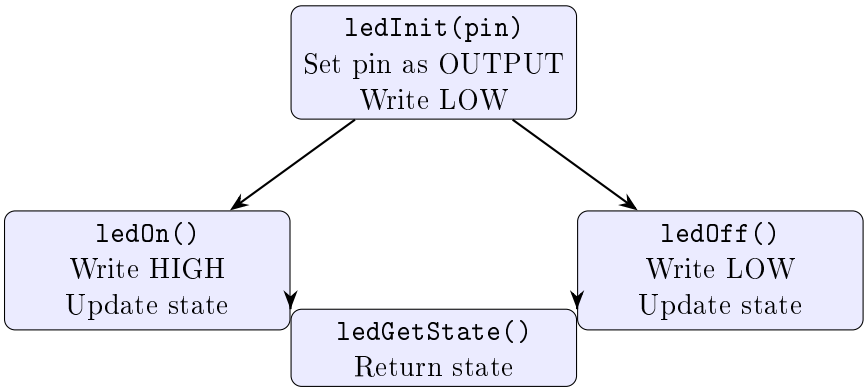

Figure 2: LED Module — Function Call Flow

2.3 Main Application Behavioral Flowchart

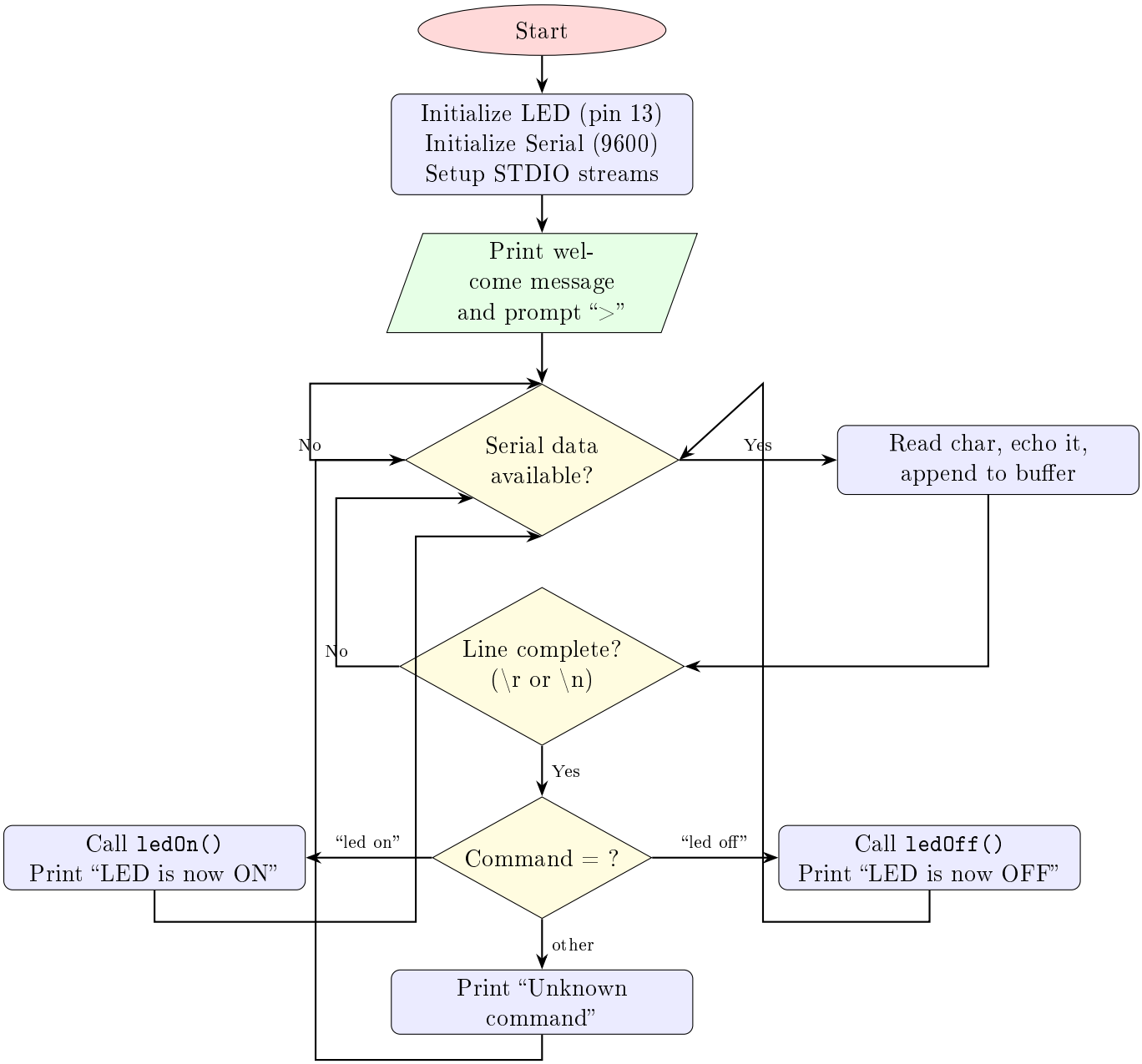

Figure 3: Main Program Flowchart

Algorithm description:

1. On power-up, `setup()` initializes the LED module (configures pin 13 as OUTPUT, sets LED to OFF) and the serial command module (opens UART at 9600 baud, redirects `stdout`/`stdin` to UART via `fdev_setup_stream()`, and prints a welcome message using `printf()`).
2. The `loop()` function continuously calls `serialCommandRead()`, which checks if serial data is available. If a character arrives, it is echoed back (using `printf("%c", c)`) and appended to an internal buffer. Backspace handling is also implemented.
3. When a newline character (`\r` or `\n`) is detected, the buffer is null-terminated and its length is returned, indicating a complete command.

4. The command string is passed to `serialCommandProcess()`, which converts it to lowercase and compares it against known commands ("led on", "led off"). The corresponding LED module function is called, and a confirmation message is printed via `printf()`.
5. If the command is not recognized, an error message is printed. The prompt ">" is displayed again, and the system returns to waiting for the next command.

2.4 Electrical Schematic

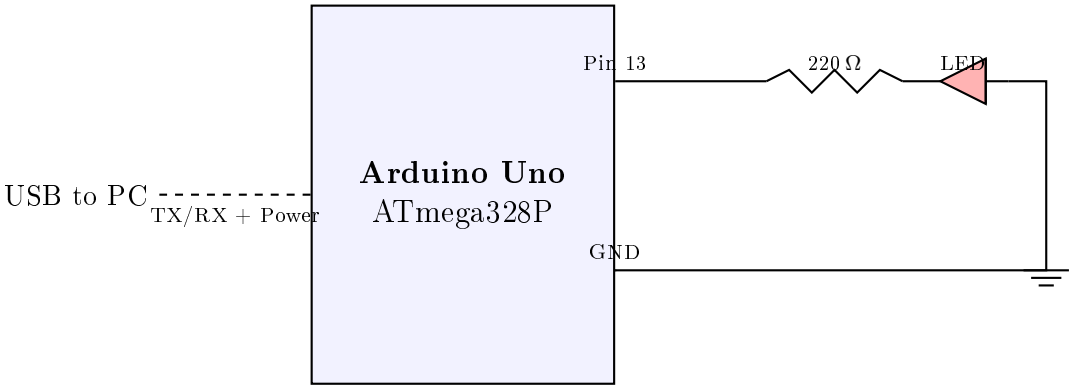

Figure 4: Electrical Schematic — LED connected to Pin 13 via 220 Ω resistor

Circuit description:

- **Pin 13 (GPIO) → 220 Ω resistor → LED anode → LED cathode → GND.**
- The 220 Ω resistor limits the current through the LED to approximately $\frac{5\text{V}-2\text{V}}{220\,\Omega} \approx 13.6\,\text{mA}$, which is within the safe operating range for both the LED and the ATmega328P GPIO pin (max 20 mA per pin).
- The **USB cable** provides both 5 V power to the Arduino and the UART data channel (TX/RX) used for serial communication with the PC.
- **Note:** The Arduino Uno has a built-in LED on pin 13 with its own resistor, so for basic testing no external components are needed.

2.5 Project Structure

The project follows PlatformIO conventions with a clean separation between interface declarations (`include/`) and implementations (`src/`):

```
Lab1/
+-- platformio.ini          # Build configuration
+-- include/
|   +-- led.h               # LED module interface
|   +-- serialCommand.h     # Serial command module interface
+-- src/
|   +-- main.cpp            # Application entry point
|   +-- led.cpp             # LED module implementation
|   +-- serialCommand.cpp   # Serial command implementation + STDIO
+-- test/
    +-- README
```

Design rationale:

- Each peripheral (LED) gets its own `.h`/`.cpp` pair. This makes it trivial to reuse the LED module in future laboratories without modification.
- The serial command module encapsulates all communication logic, including the STDIO redirection. Other modules never interact with UART directly.
- `main.cpp` contains only initialization and the main loop—it delegates all work to the modules, following the **Single Responsibility Principle**.
- **CamelCase** naming convention is used for all function names (e.g., `ledInit`, `serialCommandRead`) as recommended in the laboratory guidelines.

3 Implementation

3.1 PlatformIO Configuration

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:uno]
platform = atmelavr
board = uno
framework = arduino
```

Listing 1: platformio.ini — Project Configuration

The configuration targets the **Arduino Uno** board using the **atmelavr** platform and the **Arduino framework**. PlatformIO automatically resolves the AVR-GCC toolchain and AVR libc (which provides STDIO support).

3.2 LED Module — Interface (`led.h`)

```
#ifndef LED_H
#define LED_H

#include <Arduino.h>

void ledInit(uint8_t pin);
void ledOn(void);
void ledOff(void);
uint8_t ledGetState(void);

#endif // LED_H
```

Listing 2: led.h — LED Module Interface

The header file declares four functions that form the public API of the LED module:

- `ledInit(uint8_t pin)` — Configures the specified GPIO pin as OUTPUT and ensures the LED starts in the OFF state. The pin number is stored internally so that other functions do not need to know it.
- `ledOn()` / `ledOff()` — Set the LED state by writing HIGH or LOW to the previously configured pin via `digitalWrite()`.
- `ledGetState()` — Returns the current logical state of the LED (HIGH or LOW), useful for status queries without accessing hardware directly.

The header guard (`#ifndef LED_H`) prevents multiple inclusion.

3.3 LED Module — Implementation (`led.cpp`)

```
#include "led.h"

static uint8_t ledPin = 0;
static uint8_t ledState = LOW;

void ledInit(uint8_t pin) {
    ledPin = pin;
    ledState = LOW;
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
}

void ledOn(void) {
    ledState = HIGH;
    digitalWrite(ledPin, HIGH);
}

void ledOff(void) {
    ledState = LOW;
    digitalWrite(ledPin, LOW);
}

uint8_t ledGetState(void) {
    return ledState;
}
```

Listing 3: led.cpp — LED Module Implementation

Key implementation details:

- `static` variables `ledPin` and `ledState` are file-scoped, meaning they are not visible outside this compilation unit. This enforces encapsulation—other modules cannot accidentally modify the pin number or state.
- The `ledState` variable tracks the logical state internally, avoiding the need to read back from the hardware register (which would require accessing AVR port registers directly).
- All functions use the Arduino HAL (`pinMode`, `digitalWrite`) rather than direct register manipulation, ensuring portability across different AVR boards.

3.4 Serial Command Module — Interface (`serialCommand.h`)

```
#ifndef SERIAL_COMMAND_H
#define SERIAL_COMMAND_H

```

```h
#include <Arduino.h>

void serialCommandInit(unsigned long baudRate);
int  serialCommandRead(char *buffer, uint8_t maxLength);
void serialCommandProcess(const char *command);

#endif // SERIAL_COMMAND_H
```

Listing 4: serialCommand.h — Serial Command Module Interface

Three functions are exposed:

- `serialCommandInit(unsigned long baudRate)` — Opens the UART, redirects `stdout`/`stdin` to it, and prints the welcome message.
- `serialCommandRead(char *buffer, uint8_t maxLength)` — Non-blocking function that reads available characters from serial, assembles them into a line buffer, and returns the length when a complete line is received (or -1 if incomplete).
- `serialCommandProcess(const char *command)` — Parses and executes a command string, calling the appropriate LED module functions.

3.5 Serial Command Module — Implementation (`serialCommand.cpp`)

```cpp
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "serialCommand.h"
#include "led.h"

/* -------- STDIO redirect over UART -------- */

static int uartPutChar(char c, FILE *stream) {
    (void)stream;
    if (c == '\n') {
        Serial.write('\r');
    }
    Serial.write((uint8_t)c);
    return 0;
}

static int uartGetChar(FILE *stream) {
    (void)stream;
    while (!Serial.available()) {
        // wait for data
    }
    return Serial.read();
}

static FILE uartStream;

/* -------- Public functions -------- */

void serialCommandInit(unsigned long baudRate) {
    Serial.begin(baudRate);

    /* Redirect stdin/stdout to UART via STDIO */
    fdev_setup_stream(&uartStream, uartPutChar, uartGetChar,
        _FDEV_SETUP_RW);
    stdout = &uartStream;
```

```
    stdin  = &uartStream;

    printf("System initialized.\n");
    printf("Available commands:\n");
    printf("  led on  - Turn LED on\n");
    printf("  led off - Turn LED off\n");
    printf("> ");
}

int serialCommandRead(char *buffer, uint8_t maxLength) {
    static uint8_t index = 0;

    while (Serial.available()) {
        char c = (char)Serial.read();

        /* Handle backspace */
        if (c == '\b' || c == 127) {
            if (index > 0) {
                index--;
                printf("\b \b");
            }
            continue;
        }

        /* End of line */
        if (c == '\r' || c == '\n') {
            buffer[index] = '\0';
            uint8_t len = index;
            index = 0;
            printf("\n");
            return len;
        }

        /* Store printable character */
        if (index < maxLength - 1 && isprint((unsigned char)c)) {
            buffer[index++] = c;
            printf("%c", c);  /* echo */
        }
    }
    return -1;  /* command not yet complete */
}

static void toLowerStr(char *str) {
    for (uint8_t i = 0; str[i]; i++) {
        str[i] = tolower((unsigned char)str[i]);
    }
}

void serialCommandProcess(const char *command) {
    char cmd[32];
    strncpy(cmd, command, sizeof(cmd) - 1);
    cmd[sizeof(cmd) - 1] = '\0';
    toLowerStr(cmd);

    if (strcmp(cmd, "led on") == 0) {
        ledOn();
        printf("LED is now ON.\n");
    } else if (strcmp(cmd, "led off") == 0) {
        ledOff();
        printf("LED is now OFF.\n");
```

```
96      } else if (strlen(cmd) > 0) {
97          printf("Unknown command: \"%s\"\n", command);
98          printf("Use: led on / led off\n");
99      }
100     printf("> ");
101 }
```

Listing 5: serialCommand.cpp — Serial Command Module Implementation with STDIO Redirect

Detailed explanation of critical sections:

3.5.1 STDIO Redirection (Lines 9–27)

The core of the STDIO integration lies in two static callback functions:

- `uartPutChar(char c, FILE *stream)` — Called by `printf()` for every character to output. It writes the character to the UART via `Serial.write()`. A newline (`\n`) is automatically preceded by a carriage return (`\r`) for correct terminal display (Windows terminals expect CR+LF).
- `uartGetChar(FILE *stream)` — Called by input functions. It blocks (busy-waits) until serial data is available, then returns the received byte.

The `fdev_setup_stream(&uartStream, uartPutChar, uartGetChar, _FDEV_SETUP_RW)` macro configures a `FILE` structure with both read and write capabilities. Assigning this to `stdout` and `stdin` completes the redirection, making all subsequent `printf()` calls output through UART.

3.5.2 Non-Blocking Line Reading (Lines 47–78)

The `serialCommandRead()` function uses a `static uint8_t index` variable that persists across calls, allowing partial lines to be accumulated over multiple `loop()` iterations without blocking the MCU. Key features:

- **Echo:** Each printable character is immediately echoed back via `printf("%c", c)`, so the user sees what they type.
- **Backspace handling:** ASCII backspace (0x08) or DEL (127) removes the last character from the buffer and visually erases it from the terminal using the sequence `\b \b` (backspace, space, backspace).
- **Line termination:** Both `\r` (CR) and `\n` (LF) are accepted as end-of-line markers, ensuring compatibility with different terminal configurations.
- **Buffer overflow protection:** Characters beyond `maxLength - 1` are silently discarded.

3.5.3 Command Processing (Lines 85–102)

The `serialCommandProcess()` function:

1. Copies the input string to a local buffer and converts it to lowercase using the helper function `toLowerStr()`, making commands case-insensitive ("LED ON", "Led On", and "led on" all work).
2. Compares against known commands using `strcmp()`.
3. Calls the appropriate LED module function (`ledOn()` or `ledOff()`).

4. Prints confirmation or error messages using `printf()`.
5. Re-displays the prompt ">".

3.6 Main Application (`main.cpp`)

```cpp
/*
 * Lab 1.1 - STDIO Serial Interface: LED Control
 * MCU: Arduino Uno (ATmega328P)
 *
 * Commands (via serial terminal at 9600 baud):
 *   led on  - Turn the LED on
 *   led off - Turn the LED off
 */

#include <Arduino.h>
#include "led.h"
#include "serialCommand.h"

#define LED_PIN       13   /* Built-in LED on Arduino Uno */
#define BAUD_RATE     9600
#define CMD_BUF_SIZE  32

static char cmdBuffer[CMD_BUF_SIZE];

void setup() {
    ledInit(LED_PIN);
    serialCommandInit(BAUD_RATE);
}

void loop() {
    int result = serialCommandRead(cmdBuffer, CMD_BUF_SIZE);
    if (result >= 0) {
        serialCommandProcess(cmdBuffer);
    }
}
```

Listing 6: main.cpp — Application Entry Point

The main application is intentionally minimal:

- `setup()` calls `ledInit(13)` to configure the LED on pin 13, then `serialCommandInit(9600)` to initialize UART and STDIO.
- `loop()` calls `serialCommandRead()` on each iteration. If a complete command line has been received (return value ≥ 0), it is passed to `serialCommandProcess()` for execution.
- All constants (`LED_PIN`, `BAUD_RATE`, `CMD_BUF_SIZE`) are defined as macros for easy modification.

4 Results

4.1 Build Output

The project compiles successfully using PlatformIO:

```
> pio run
Processing uno (platform: atmelavr; board: uno; framework: arduino)
...
```

```
Building in release mode
Compiling .pio/build/uno/src/led.cpp.o
Compiling .pio/build/uno/src/main.cpp.o
Compiling .pio/build/uno/src/serialCommand.cpp.o
Linking .pio/build/uno/firmware.elf
Checking size .pio/build/uno/firmware.elf
Building .pio/build/uno/firmware.hex
================================= [SUCCESS] ================================
```

Listing 7: Build Output

4.2 Serial Monitor Interaction

After uploading the firmware to the Arduino Uno and opening the serial monitor at 9600 baud, the following interaction is observed:

```
System initialized.
Available commands:
  led on  - Turn LED on
  led off - Turn LED off
> led on
LED is now ON.
> led off
LED is now OFF.
> Led On
LED is now ON.
> hello
Unknown command: "hello"
Use: led on / led off
> led off
LED is now OFF.
>
```

Listing 8: Serial Terminal Session Example

Observations:

- The system prints a welcome message and a list of available commands on startup.
- The "`led on`" command successfully turns the LED on, and a confirmation message is displayed.
- The "`led off`" command successfully turns the LED off with confirmation.
- Commands are **case-insensitive**: "Led On" is accepted and processed correctly.
- Unrecognized commands produce a helpful error message suggesting valid options.
- The prompt "$>$" is re-displayed after each command, providing a user-friendly interactive experience.

4.3 Hardware Demonstration

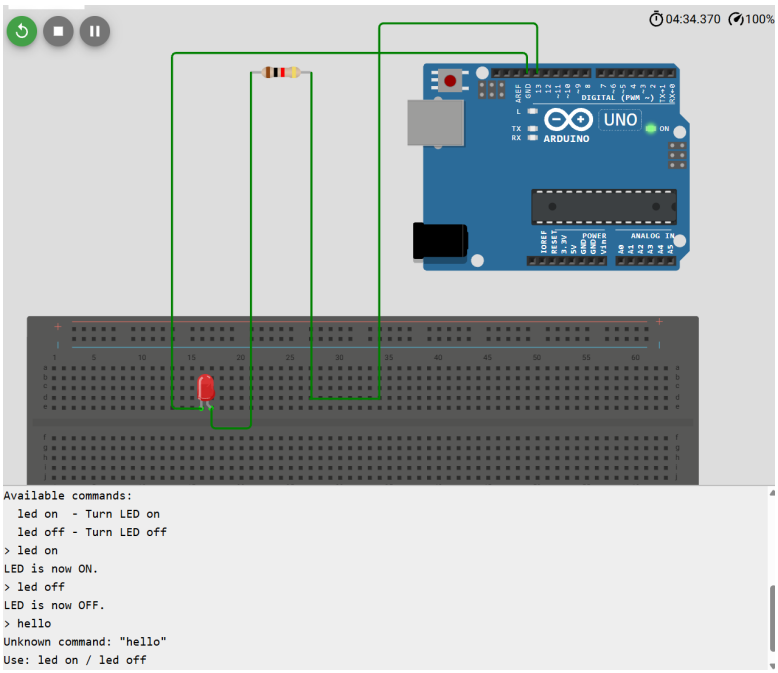

Figure 5: Arduino Uno with LED ON (Simulation)

5 Conclusions

5.1 Performance Analysis

The implemented system meets all laboratory requirements:

- **STDIO usage:** All text output uses `printf()` and input is handled through the redirected `stdin` stream, satisfying the requirement to use the STDIO library instead of Arduino-specific functions.
- **Command processing:** The system correctly interprets "led on" and "led off" commands and responds with appropriate confirmation messages.
- **Modular architecture:** The LED peripheral is encapsulated in its own module (`led.h`/`led.cpp`) making it directly reusable in subsequent laboratory works without modification.
- **CamelCase naming:** All function names follow the CamelCase convention as recommended.

5.2 Limitations and Improvements

- **Single LED:** The current implementation supports only one LED. The module could be extended to support multiple LEDs by passing the pin number or an LED identifier to `ledOn()`/`ledOff()`.
- **Blocking input:** The `uartGetChar()` function uses busy-waiting. In a more complex system, interrupt-driven UART or a RTOS approach would be preferable.
- **Command parser:** The current string comparison approach is simple but does not scale well. A more robust solution could use a command table with function pointers.

- **Memory usage:** The use of `printf()` on AVR adds approximately 1.5 KB to the firmware size due to the formatting library. For extremely memory-constrained applications, lighter alternatives exist.

5.3 Real-World Applicability

The STDIO-over-UART pattern demonstrated in this laboratory is widely used in embedded systems for:

- **Debugging:** `printf`-style logging is the most common debugging technique in embedded development.
- **Command-line interfaces (CLI):** Many industrial devices, routers, and IoT gateways expose a serial CLI for configuration and diagnostics.
- **Data logging:** Sensor data can be formatted and transmitted via `printf()` for analysis on a PC.

6 Note on AI Tool Usage

During the preparation of this report and the development of the source code, the author used **GitHub Copilot** (powered by a large language model) as an AI-assisted coding tool within Visual Studio Code. Specifically, Copilot was used for:

- Generating the initial modular project structure and boilerplate code.
- Assisting with the STDIO redirection implementation (`fdev_setup_stream` setup).
- Drafting sections of this report.

All AI-generated content was reviewed, validated, and adjusted by the author to ensure correctness and compliance with the laboratory requirements.

7 Appendix — Source Code

The complete project source code is available in the GitHub repository:

https://github.com/Tirppy/si-course-repo/tree/main/Lab1

The repository contains the full PlatformIO project with the following structure:

```
Lab1/
+-- platformio.ini          # Build configuration
+-- include/
|   +-- led.h               # LED module interface
|   +-- serialCommand.h     # Serial command module interface
+-- src/
|   +-- main.cpp            # Application entry point
|   +-- led.cpp             # LED module implementation
|   +-- serialCommand.cpp   # Serial command + STDIO redirect
```

8 Bibliography

[1] **ATmega328P Datasheet**, Microchip Technology Inc., 2018.
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcont
Datasheet.pdf

[2] **AVR Libc Reference Manual** — `<stdio.h>`: Standard IO facilities.
https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html

[3] **PlatformIO Documentation** — Arduino Uno Board Configuration.
https://docs.platformio.org/en/latest/boards/atmelavr/uno.html

[4] **Arduino Reference** — `Serial` class documentation.
https://www.arduino.cc/reference/en/language/functions/communication/serial/

[5] **Arduino Reference** — `digitalWrite()` function.
https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/